FEATURES

FPGA Design: A Fresh Take

An Alternative to MCUs

Although FPGAs are a well-established technology, many embedded systems developers—particularly those used to the microcontroller realm—have never used them. Here, Faiz takes a fresh look at FPGAs for those new to designing them into their embedded systems.

By
Faiz Rahman
Ohio University

PHOTO 1
FPGAs consist of a large number of digital logic blocks that can be interconnected, through programmable routing on the chip itself.

Miniaturization and component integration have been driving the evolution of electronics for a hundred years. Vacuum tube electronics gave way to solid-state components which, in turn, evolved into integrated circuits (ICs). Monolithic ICs continue to advance through ever-increasing levels of integration. With minimum transistor feature sizes fast approaching 10 nm, hundreds of millions of transistors can be packed onto a single chip no bigger than a thumbnail. While the majority of advanced commercial ICs feature fixed circuits performing one specific function, a sizeable fraction of digital ICs enables users to configure the internal circuitry in almost any fashion they like. These programmable logic ICs—more typically called FPGAs (**Photo 1**)—consist of a large number of digital logic blocks that can be interconnected through programmable routing on the chip itself. Their programming, or more appropriately, configuration, is done through circuit description languages. Once configured, an FPGA behaves in the way its user intended—essentially acting as a user-designed digital IC.

If you have never used this type of device then you are missing out on some of the most astounding technological developments of our times—comparable to the microcontroller revolution of the 1980s and 90s. For many—particularly embedded engineers who have traditionally focused on MCU-based designs—FPGAs may have seemed too complex to integrate into their designs. In recent years, FPGAs have become much easier to use with powerful tools supporting them. This article will, hopefully, remove any barriers that might come in the way of using this thrilling technology. First, we'll look at the basics of programmable logic ICs: their structure, function and capabilities. Then we'lll take explore how you can easily configure them to perform almost any digital circuit function.

DIFFERENT PHILOSOPHY

The philosophy behind FPGAs is quite a bit different from that of microcontrollers. In the latter, a set of instructions, called a program, is executed serially to perform desired actions. Complex calculations running on microcontrollers may take a significant amount of time to execute because thousands of simple instructions may need to be performed one after the other. In contrast, no software runs on FPGAs. Instead, their internal circuitry is configured—as explained later—to perform a specific function. This is a pure hardware approach, which is typically much faster than carrying out the same action through the execution of simpler actions under program control. Tasks carried out by FPGAs can run much faster than when the same

tasks are performed by microcontrollers.

The technology that allows circuits to be electronically configured on a chip is based on connecting two conducting traces through a transistor switch. Actually, it is more than just a transistor as it is more like a single SRAM cell. If the bit stored there is 1 then the interconnection is enabled—otherwise, the two conducting lines remain electrically disconnected from each other. By having an SRAM cell at the intersection of a grid of conducting lines, it is possible to build a routing network. Now imagine that some general-purpose logic gates are placed inside every block of this grid and you have a basic programmable logic IC.

The first such devices appeared in the 1980s and simply consisted of arrays of AND and OR gates that could be connected to each other through a programmable routing network. While such devices are still available, the most evolved modern form of programmable logic devices are FPGAs. Their main distinction from the simpler programmable logic devices of the past is in their use of so-called logic elements (LEs)— also called logic blocks (LBs) and logic array blocks (LABs). LABs are the smallest unit of functionality in FPGAs. Each LAB, basically, consists of a look-up table (LUT) and a flip-flop (FF). A LUT is, typically, a 4-to-1 multiplexer with inputs tied to known logic levels. This arrangement acts as an AND-to-OR combinational logic element with the output determined by the selection input fed to the multiplexer.

The LUT output can go to the output of the LAB, directly or through a clocked D-type FF. The latter ensures that sequential circuits could also be implemented. Generally, real LABs are quite a bit more complex than this description (**Figure 1**) but the main idea of LUTs followed by FFs is almost universal. The LABs are arranged in a two-dimensional array with interconnection wiring, controlled by SRAM cells, running between them. In this context, the interconnection wiring is usually called the "FPGA fabric" (**Figure 2**). This pattern is so cleverly laid out that any LAB can share its output with any other LAB on the chip. Real FPGAs have a few other components also integrated on the chip. These might include phase-locked loops (PLLs) for clock generation, embedded memory blocks, embedded multipliers and so on.

EMBEDDING CIRCUITRY

So how can a digital circuit be placed inside an FPGA? The secret lies in configuring LABs and their interconnection pattern. By placing specific bit patterns at LUT inputs, the LABs are made to mimic simple combinatorial AND-OR type logic circuits. Then by placing

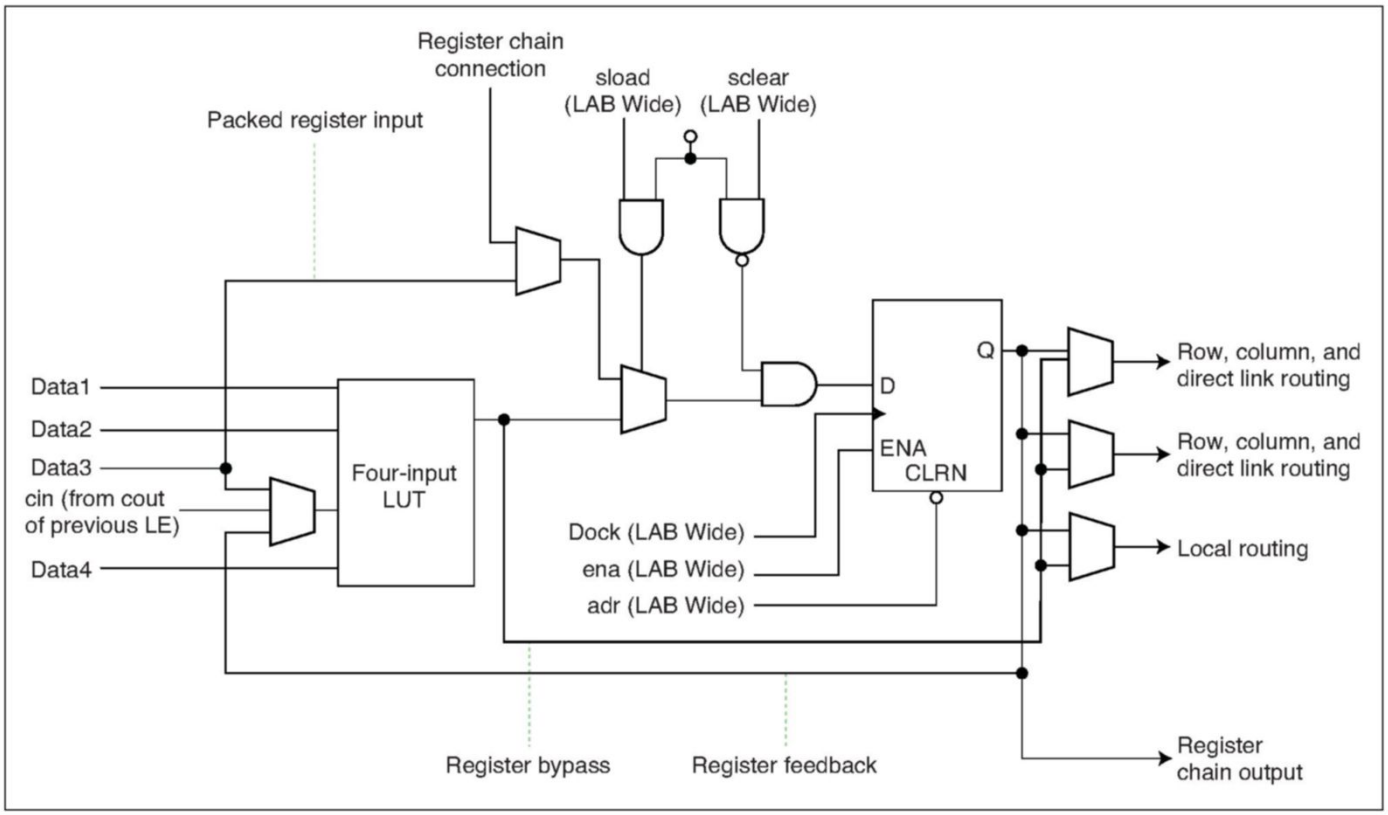


FIGURE 1
This one logic array block (LAB) in an Altera MAX 10 FPGA shows LUT, FF and various inter-LAB connections.

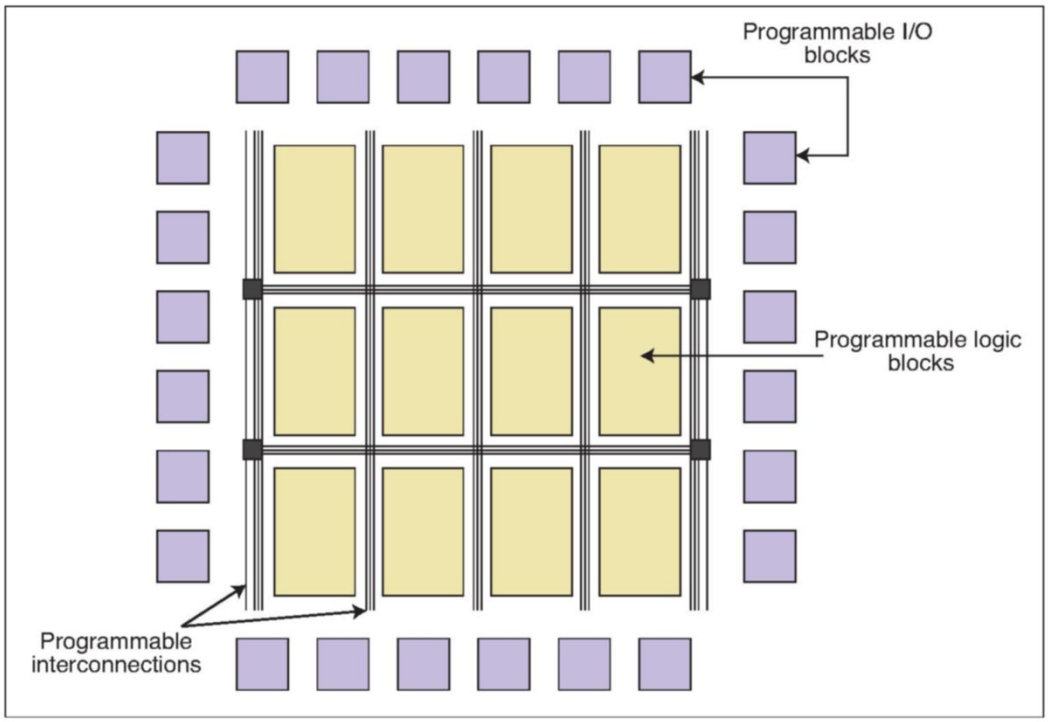


FIGURE 2
Shown here is the distribution of programmable logic array blocks amid interconnection wiring in a typical FPGA. I/O blocks that connect to I/O pins are also seen.

another specific bit pattern with their interconnection SRAM cells, the LABs are appropriately connected to each other to take on the function of a simple or complex digital circuit. That's all there is to it.

The bit pattern needed for this purpose is generated by an FPGA application development software much as the object file needed for a microcontroller is generated by a microcontroller application development software. Whereas the file to be downloaded into a microcontroller is, usually, a .hex file, the file that is downloaded to an FPGA is called a .sof file ("sof" stands for SRAM object file). Configuring an FPGA is essentially the same as placing a bit pattern into the cells of an SRAM array. Like SRAMs, FPGAs are also volatile—removing the power erases the configuration bit pattern, turning the FPGA back into a blank slate. This makes it necessary to use an accompanying flash memory that can feed the FPGA with its configuration file each time the system's power is turned on.

Serial flash memories performing this role are called configuration memories. On power up, FPGAs look to download their configuration data from a nearby configuration flash memory. This process only takes a few milliseconds and then the FPGA starts behaving in the intended manner. A small minority of modern FPGAs have an on-chip flash memory block where configuration data can be stored. On power up, such FPGAs tend to read the internal flash memory and use its contents to program the SRAM cells in the FPGA fabric. This feature removes the need for a separate flash memory device to hold configuration data. Altera's MAX 10 family and Microsemi's IGLOO family (**Photo 2**) are examples of this type of flash FPGA.

MANY FPGA FEATURES

Beyond being just a collection of uncommitted logic blocks connected by a configurable interconnection system, modern FPGAs have a host of other useful features. Unlike microcontrollers, FPGAs boast a vastly greater number of I/O pins. Even small FPGAs have close to a hundred I/O pins while larger devices can have several hundred of them. When combined with hundreds to hundreds of thousands of LABs on a single chip, it's clear that powerful I/O-rich systems can be created in FPGAs. FPGAs are, generally, capable of running at much higher clock frequencies compared to small microcontrollers—100 MHz to 400 MHz is typical for FPGAs. And FPGA circuits perform operations in real time. With all that in mind, it's a given that functions implemented in FPGAs operate extremely fast compared to traditional processors.

It's also important to keep in mind that nothing prevents a designer from putting multiple, related or completely separate circuits on the same FPGA. That way the functions of several separate digital ICs can be consolidated into a single FPGA. Such logic consolidation is a big reason for the popularity of FPGAs in today's consumer, industrial and military electronics. In fact, space and defense sectors are such heavy users of FPGAs that semiconductor manufacturers

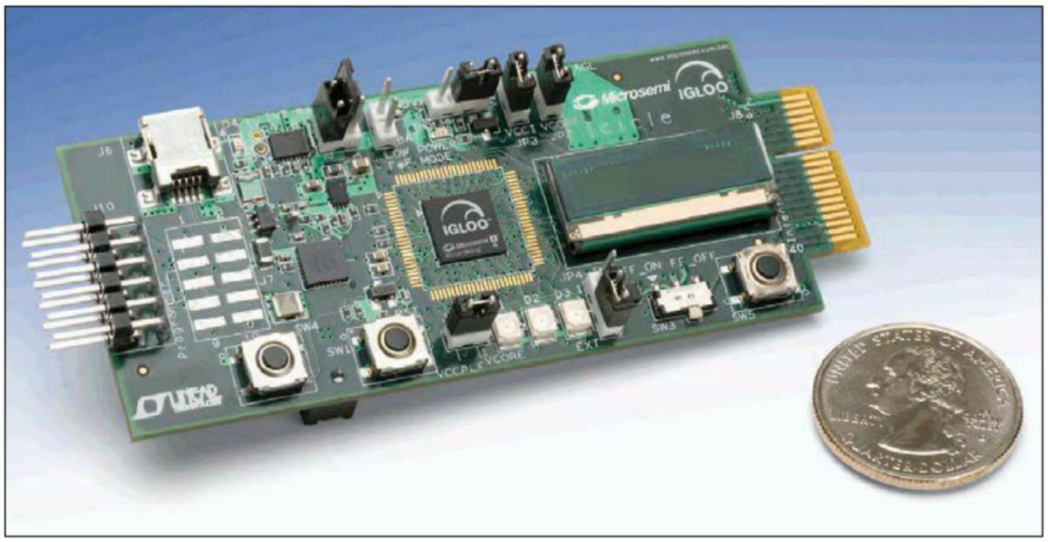


PHOTO 2
Microsemi IGLOO flash FPGA on an FPGA development board

make special radiation-tolerant versions of standard devices for those markets. Most commonly available FPGAs contain a number of other useful features such as PLLs for generating almost any frequency on chip, embedded memory blocks, fixed- (and even floating point) multipliers, analog-to-digital converters (ADCs) and so on. High-end FPGAs also feature integrated high-speed serial data transceivers.

Microcontrollers are definitely easier to use when compared to FPGAs and there is no good reason not to use microcontrollers where they fit well in an application. However, FPGAs come into their own for certain applications. These include operations that require lots of I/O pins (**Photo 3**) such as scanning arrays in cameras and displays. Complex algorithms for digital signal processing can also benefit from the real-time in-hardware processing capabilities of FPGAs. As a result, high-definition (HD) and ultra-high-definition (UHD) 4K and 8K display processing system designs make heavy use of FPGAs. Military sensor signal data processing is also very reliant on the use of high-end FPGAs.

For hobbyists, FPGAs offer a number of benefits. Perhaps the foremost is the ability to have a chip carry out your own customized task. Even if a standard IC is not available to perform a certain operation, an FPGA could be configured to carry out that task. From generating multiple precisely synchronized clocks to control CCD chips to performing desired mathematical operations on a data stream, FPGAs can be called on to perform one or multiple tasks quickly and efficiently. They are also good at communication protocol conversion. It is relatively easy to use FPGAs to convert between USB, Ethernet and serial data communications. FPGAs also make great glue-logic devices—interfacing a processor with various memory and I/O devices. There is basically no end to what can be achieved with FPGAs because they offer sheer unconstrained digital data handling capability.

GETTING STARTED WITH FPGAs

So, what is needed to start using FPGAs? The answer is hardware description languages (HDLs)—used to define a digital system. Once the desired digital function is described in the form of HDL code, the code can be compiled by an HDL compiler to generate a net list specifying the interconnection of LABs in an FPGA. This net list is essentially just a pattern of bits that that can program the FPGA's internal SRAM cells to make it assume the identity of a custom digital device. The output .sof file from the HDL compiler can be downloaded either directly to an FPGA or to a flash memory device connected to the FPGA. In the former case, the FPGA will need to do a download every time it is turned on. But in the latter case it will load its configuration file automatically from the configuration memory on power up. Direct download into an FPGA is used during configuration code development

PHOTO 3
These top and bottom views of Microsemi's PolarFire FPGA show a profusion of I/O pins.

and debugging. Once finalized, the external or internal flash memory—where available—is programmed with the HDL compiler-generated bit stream to create a normal functioning system.

Obviously learning an HDL is the key to using FPGAs and any serious user of FPGAs has to spend some effort at learning to code in at least one HDL. There are several of these around but two—VHDL and Verilog— are by far the most common HDLs used by 95% of FPGA users around the world. Here is the fun part: An HDL enables a developer to describe the desired system in a high-level form without worrying about its exact implementation in terms of logic gates. That's the function of the HDL compiler that takes the design code and generates a circuit design in the form of a net list. The HDL coder also doesn't need to understand too much about the architectural details of the FPGA being used. The compiler generates the best circuit configuration for the target device in an internal process called "place and route." All the application developer has to do after a .sof has been successfully generated is to assign some of the FPGA's many I/O pins to carry signals into and out of the device. After this pin assignment, the HDL code is compiled once again and then the final .sof is ready for use with the FPGA.

HDL LANGUAGE FEATURES

We will take a look at VHDL in some detail later, but some prominent features of this language can be discussed here. The name, VHDL, contains an abbreviation within an abbreviation. It stands for VHSIC hardware description language, with VHSIC standing for very high speed integrated circuit. It was developed by a consortium of several electronics companies during the 1980s at the behest of the Department of Defense (DoD). The reasons behind this development are historical and don't concern us here. Later the language was standardized by the Institute of Electrical and Electronics Engineers (IEEE) and it now exists as a highly uniform language. With its genesis in the defense community, it's not surprising that VHDL borrowed most of its syntax from another DoD-sponsored language: Ada. Meanwhile the Ada programming language is itself a variant of the Pascal programming language and thus the syntax of VHDL strongly resembles Pascal (**Listing 1**).

```
pwm_process : process( clk_1hz)
    begin
    if (rising_edge(clk_1hz)) then
        if (t_on = 0) then pwm_signal <='0';
    elseif((count <= t_on) and (count < top_cnt)) then
                pwn_signal <= '1';
                    count <= count + 1;
                elseif((count > t_on) and (count < top_cnt)) then
                    pwn_signal <= '0';
                    count <= count + 1;
                else pwm_signal <= '0'; count <= 0;
                                end if;
                end if;

                for i in 7 downto 0 loop
                    pwn_leds(i) <= pwm_signal;
                end loop;
end process pwm_process;
```

LISTING 1
This portion of VHDL code shows strong a resemblance to Pascal and Ada.

Superficially VHDL looks like Pascal, but that's where the similarity ends. When you write VHDL code—or for that matter, code for any other HDL, such as Verilog—you are writing the description of a task that needs to be performed through digital means. This is a behavioral description rather than a structural description. In other words, you describe the desired behavior of your system instead of what the actual circuit is. The latter is created by the HDL compiler. We call this process "inferring a circuit." This is an extremely powerful concept. A designer simply concentrates on the task at hand, without worrying about the digital circuit needed to get the job done. Once a proper VHDL or Verilog description has been coded, the compiler will take it and come up with an appropriate interconnection of the LABs inside an FPGA to accomplish the task. That's all there is to it. The net list .sof file generated by the compiler—after pin assignments—can be downloaded to an FPGA or its configuration memory device and the job is done.

A piece of VHDL or Verilog code that generates a functioning digital circuit inside an FPGA can be re-used as part of some other bigger application. Many such re-useable, general-purpose code blocks are often made available as a library. In this form, the code is called intellectual property (IP). IP for such tasks as floating-point multiplication, video encoding/decoding, Ethernet MAC and so on are commonly made available by FPGA manufacturers. There is also a significant market in third party-developed IP that is available for sale and licensing.

USING FPGAs

Now that we've explored the basic concepts about programmable logic, next we'll examine how FPGAs are actually used. This takes us into hardware description languages (HDLs) used for configuring programmable logic devices and the hardware/software environment

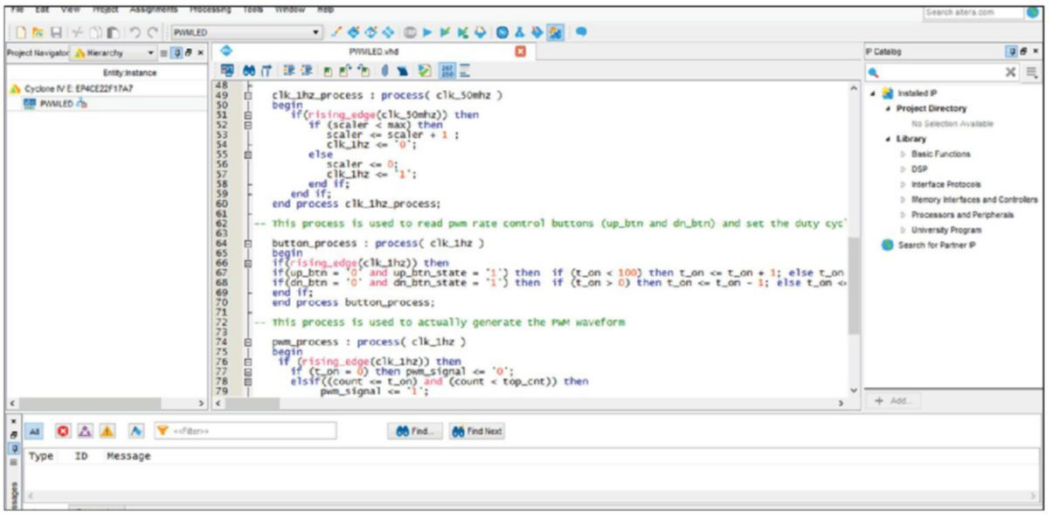


PHOTO 4
VHDL code strongly resembles PASCAL and Ada. Shown here is VHDL code open in the editor window of Intel's Quartus Prime software.

used for application development. In order to use FPGAs, you need an FPGA device on a PCB—usually on a development board with USB programming interface—and an HDL compiler. This is a combination that's very similar to what's needed when working with microcontrollers. HDL compilers—also called FPGA application development software—can be freely downloaded from FPGA manufacturers' websites.

Each manufacturer offers different software, but they all have more-or-less the same basic functionality. All of them allow entry of HDL code in either Verilog or VHDL, syntax and semantics checking, compilation into a net list for a targeted FPGA device and various reports on FPGA resource utilization, power consumption, timing verification and so on. Manufacturers' websites also offer training documentation and videos on how to use their development environment. Once, the basics of either Verilog or VHDL is learned, it becomes very easy to use any one of these compilers to configure an FPGA to perform one or more desired functions.

As mentioned earlier, HDL code looks very similar to code written in ordinary software programming languages. VHDL code, in particular, bears a strong resemblance to Pascal and Ada. This is seen in **Photo 4** which shows VHDL code open in the editor window of Altera (Intel) Quartus Prime software. In spite of its similarity to conventional software programming languages, an HDL coder has to keep in mind that the code being written is not step-by-step instructions to perform a task but the description of a digital system. The code describes the behavior of a digital system, leaving the exact circuit implementation to the software. All that said, the designer has to keep in mind certain limitations imposed by both the HDL and the FPGA technology being used, in order to write code that will be acceptable to the compiler.

Errors are flagged by the compiler in much the same way as when designing a microcontroller application. Once errors are fixed and a .sof file has been successfully generated, it can be downloaded to the FPGA through a JTAG programming interface. This is usually implemented in such a way that all a user sees is a simple USB connection between a PC and an FPGA development board. One the design is working on the FPGA, its function can be validated. Later, the design can be placed in a flash-based configuration memory so that it's automatically downloaded into the FPGA each time the power is turned on.

SIMPLICITY OF VHDL

Now let's take a deeper look at VHDL to see how the language lets you express desired tasks in a simple manner. To begin with, all HDLs differ from software programming languages in that they allow what is called 'concurrency'. This simply implies parallelism. When two or more statements are written in an HDL—one after the other—instead of being executed in that order they are all executed together at the same time. This concept may be a little hard to grasp if it is new to you. But if you think about, it's how things should be. After all hardware works in parallel in real time, so a language used to describe it should also have this capability. With that in mind, writing A = X1 AND X2; followed by B = Y1 AND Y2; will generate circuitry equivalent to two two-input AND gates working side by side. It is possible to have other types of statements that do operate sequentially, in the order of their appearance. In VHDL these statements have to be placed inside a construct called a "process".

There can be any number of processes in a piece of VHDL code. While statements inside processes operate sequentially—as in

software programming languages—processes themselves execute concurrently with each other. That means a process can be used to model a circuit that is more complicated than can be expressed with a single statement. Another essential feature HDLs have is the ability to represent electrical signals, in addition to variables and constants. Again, this is only to be expected. In VHDL a special class of data objects called a signal is used for this purpose. A standard logic data type is also made available to represent logic 1s and 0s (and the high impedance state 'Z'). A standard logic signal—with a name chosen by the user—can be used to represent a single bit. Using a signal named "read_strobe," a declaration for such a signal in VHDL code looks like this:

```
signal read_strobe : std_logic;
```

A multi-bit vector (a signal bus) could be declared as follows:

```
signal memory_address : std_logic_vector
(7 downto 0);
```

Assignments can be made to these signals in several ways, including the following:

```
read_strobe <= '1';  -- assigns value to
bit signal read_strobe
memory_address <= "10101010"; -- assigns
value to byte signal memory_address
```

Note that a double dash "--" is used to start a comment for code readability. The rest of the line after the "--" is ignored by the VHDL compiler.

In VHDL a digital system is described using two separate but interlinked design units. The first, called an 'entity', describes the desired system's I/O interface or port—such as all signals coming in and going out of the system. This part regards the system as a black box and pays no attention to the function or structure of the system. Here is a simple entity description for a pulse width modulation (PWM) circuit that reads two input push-buttons to increase or decrease PWM duty cycle and uses the waveform to drive eight LEDs. It also accepts a clock waveform from an external oscillator. An entity can have any name chosen by the coder—here it is called PWMLED. 'in' and 'out' signify direction of I/O—whether it is coming into the system or leaving it respectively.

```
entity PWMLED is
   port(
        clk_50mhz : in std_logic ;
        up_btn : in std_logic;
        dn_btn : in std_logic;
        pwm_leds : out std_logic_vector(7
downto 0)
   );
end PWMLED;
```

VHDL ARCHITECTURE UNIT

Next comes the 'architecture' unit. This is where you describe the system's functionality. An entity architecture pair can be compiled by a VHDL compiler to generate a circuit net list as a .sof file. HDLs contain constructs that are similar in software programming languages—loops, assignment statements, conditional expressions and so on. However, in the case of an HDL, these statements reflect the way the desired system is supposed to behave. The HDL compiler makes use of the statements to work out the circuit configuration needed to accomplish the desired functionality. As an example, a D-flipflop is described in VHDL using the following code:

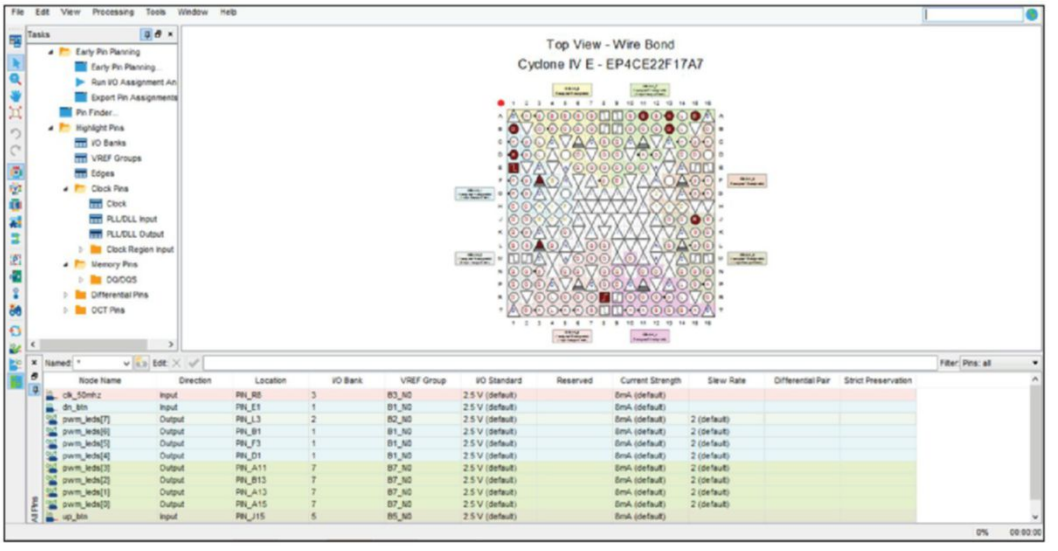


PHOTO 5
The Quartus Prime FPGA software has a utility called Pin Planner. Shown here, it provides a graphical view of the FPGA chip's pins as well as a table of all I/Os, their signal directions, logic levels and pin assignments.

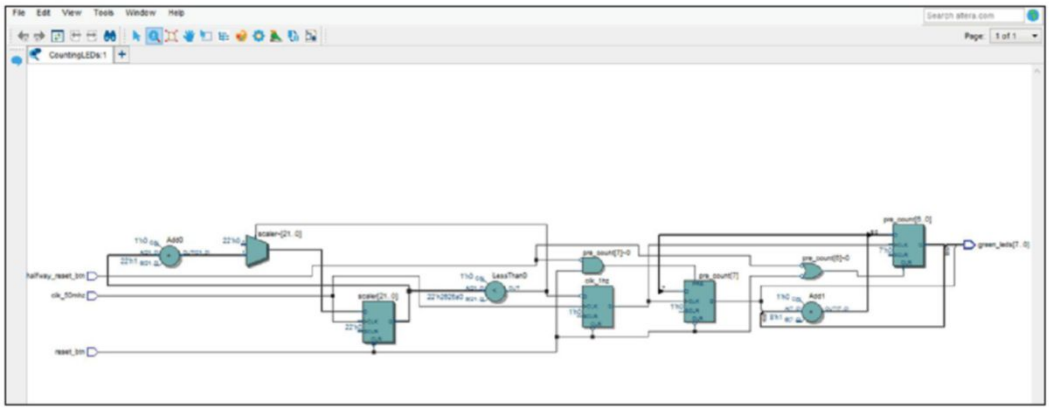


PHOTO 6
You can view the circuit configuration that will be implemented inside an FPGA by using the schematic viewer built into HDL compilers. A part of such a schematic layout shown here.

```
library ieee;
use ieee. std_logic_1164.all;

entity D_FlipFlop is
      PORT( D, CLOCK: in std_logic;
      Q: out std_logic);
end D_FlipFlop;

architecture behavioral of D_FlipFlop is
begin
     process (CLOCK)
          begin
             if (rising_edge(CLOCK)) then
              Q <= D;
          end if;
     end process;
end behavioral;
```

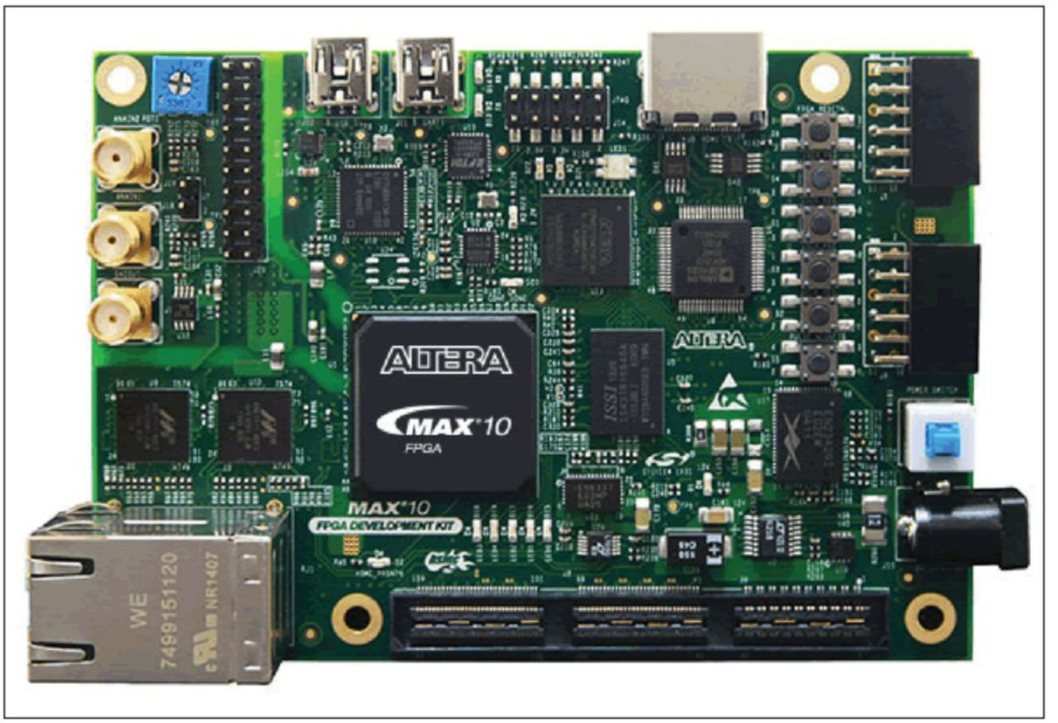
As the code shows, the first two lines direct the compiler to include an IEEE standard library that contains vital definitions. These compiler directives are present in all HDL code. Next comes an entity declaration that spells out the ports (I/Os) along with their directions and data types. Finally, the actual behavior is contained in the architecture definition. Here it has been given the name "behavioral," but it is entirely up to the code writer which names are chosen for entity and its associated architecture.

Inside the architecture shown, the main action is described within the process construct that simply acts as a container for a specific behavior. Statements within a process block only take effect if there is a transition—change in logic level—on any signal listed in the bracket after the keyword process. This is called the process' sensitivity list. In the example shown, whenever the clock signal transitions the process becomes activated. However, due to the conditional if statement, the action Q <= D takes place only when the clock signal transition is from logic 0 to logic 1 (the rising edge). This assigns the value of 1-bit input signal D to 1-bit output signal Q. This then specifies the action of a D-flipflop. If D and Q are specified as multi-bit vectors—such as `std_logic_vector (7 downto 0)`—then this same code will serve as an 8-bit register. The purpose of listing this little code snippet is to show that FPGA configuration languages are no more difficult to learn and use than conventional software programming languages, such as Pascal or C.

Learning an HDL is similar to learning a software programming language because they have similar constructs—conditional statements, loops and so one. The big difference is HDLs are used in the context of hardware (circuit) generation. With that in mind, a loop, for instance, will result in the generation of a number of similar circuit blocks instead of resulting in the repeated execution of one or more statements. It takes some effort to learn either Verilog or VHDL but this is easy to do. There are lots of free resources available on the web, as well as the availability of good books and helpful videos.

As stated before, HDL code can be compiled to produce an FPGA configuration file. Once this .sof net-list file has been created, you can assign actual physical device pins to the various inputs and outputs. In Intel's Quartus Prime FPGA application development software, this is done with a utility called Pin Planner, as seen in **Photo 5**. It shows a graphical view of the FPGA chip's pins as well as a table of all I/Os, their signal directions, logic levels and pin assignments. When these assignments have been made, a final compilation is performed to generate a .sof file with the chosen pin assignments. This output file can then be downloaded to the FPGA through a JTAG interface running over a USB link from a PC. Job done. The FPGA then behaves as a digital system of your own design. And it can be connected to other components through assigned pins. It is also possible to actually see the circuit configuration that will be implemented inside an FPGA. To do this, you can use the schematic viewer built into HDL compilers. A part of such a schematic layout is in **Photo 6**.

PHOTO 7
Easy-to-use FPGA development boards are available from a variety of board manufacturers. This is an FPGA development board based on Intel PSG's MAX 10 FPGA.



FPGA HARDWARE

On the hardware side, FPGAs are manufactured by a handful of semiconductor companies. The main ones are: Xilinx, Altera (now Intel PSG division), Lattice Semiconductor, Microsemi and Quicklogic. Each of these companies has its own line of FPGA devices, ranging from relatively small, low-cost ICs to very large devices with hundreds of thousands of LABs and hundreds of I/O pins. It is no surprise that FPGA prices vary widely from just a few dollars each for small devices to hundreds and even thousands of dollars for premium chips. The websites of FPGA manufacturers provide valuable insights into latest FPGA technologies, IP availability and application trends.

Many easy-to-use FPGA development boards are available from a variety of board manufacturers. All of them contain an FPGA together with other useful support devices—such as memories, ADCs, clock oscillators and so forth. These development boards range in price from less than a hundred dollars to thousands of dollars. **Photo 7** shows an FPGA development board based on Intel PSG's MAX 10 FPGA. Development boards generally come with detailed documentation and code examples on how to use them and exploit their many capabilities. Less expensive boards are ideal for hobbyists to learn the use of FPGAs and to use them in embedded projects. More advanced users can find higher-end boards for learning to use FPGAs with Gbit Ethernet, HDMI interfaces and high-speed data transmission over differential transceivers.

In addition to a development board, FPGA system developers also need a software tool for writing and compiling HDL code. HDL compilers can be easily downloaded for free from the websites of all FPGA manufacturers. Xilinx offers its ISE and Vivado development suites. Lattice Semiconductor provides Diamond HDL compiling tool. And Intel PSG makes its Quartus Prime software available to all FPGA application developers. These development tools can be used for compiling either VHDL or Verilog code. You can also enter gate-level schematic digital circuit diagrams and let the tools produce a net list file from it. It's even possible to mix HDL code and schematic circuit diagrams in the same project.

Once some code has been debugged, ,tested and found to operate well, it can be

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

RESOURCES

Intel PSG (formerly Altera) | www.altera.com

Lattice Semiconductor | www.latticesemi.com

Microsemi | www.microsemi.com

Quicklogic | www.quicklogic.com

Xilinx | www.xilinx.com

represented as a block diagram and used as a component in future projects. This kind of code reusability is a great feature that makes it possible to build large projects incrementally by breaking them into simpler components that are individually designed, one at a time. Such user-developed IP components are simply connected together in a graphical fashion to define the complete system. **Figure 3** shows part of a system defined in the Quartus Prime tool as an interconnection of separately developed IP blocks.

For beginners, it is best to get hold of any good FPGA development board and begin by writing simple Verilog or VHDL code and then compiling and downloading it to the FPGA. The compiler will flag syntax errors and even logic errors, if any, which is a useful way of learning an HDL. Once enough practice has been achieved, you can go on to write larger and more useful pieces of code that actually accomplish real functions. For instance, you can start by writing simple behavioral code for counters, 7-segment display drivers, PWM generators and so on. Then you can move on to more advanced code for such applications as an SD card reader/writer, a VGA display controller, a real-time clock, an Ethernet communication module and so on.

EASY TO GET STARTED

This article has provided a broad overview of FPGA technology. Hopefully this will be sufficient to convince those who are new to FPGAs, that they are both fun and easy to use. All that needs to be done to start using FPGAs is to learn an HDL (either Verilog or VHDL), obtain one of the numerous FPGA development boards available and install an HDL compiler compatible with the chosen FPGA. Several books and online resources provide simple introductions to VHDL and Verilog. One month of learning should be enough to get into serious FPGA application development. Aside from manufacturer's websites, sites such as www.fpga4fun.com also provide helpful tips and advice, as well as several introductory and mid-level projects for FPGA users to build. Large numbers of helpful videos are also available on YouTube. It doesn't take long to get up to speed with using FPGAs. After gaining some familiarity with FPGA configuration concepts, system designers can graduate to some really interesting projects.

ABOUT THE AUTHOR

Faiz Rahman has more than 20 years of experience in electronics and optoelectronics. He holds a PhD in Electrical Engineering from Imperial College, London, and is currently a member of Electrical Engineering faculty at Ohio University. Faiz can be reached at rahmanf@ohio.edu.

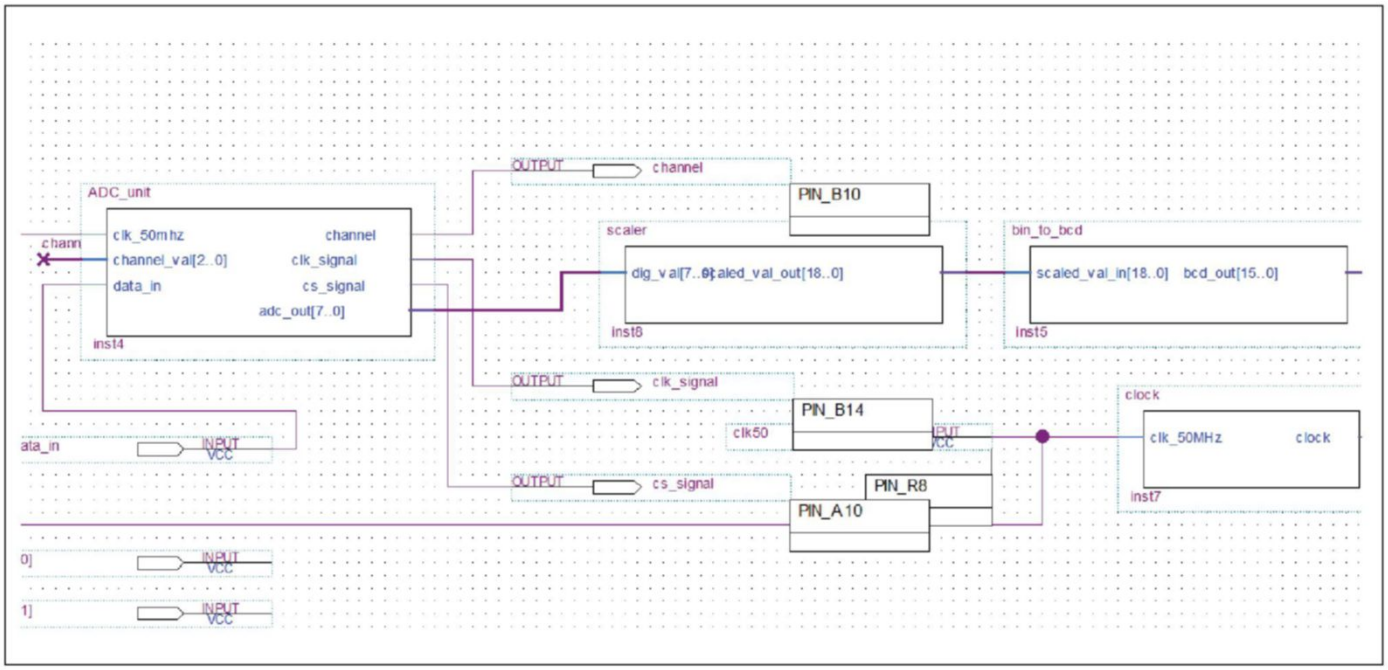


FIGURE 3

Using VHDL, it's possible to build large projects incrementally by breaking them into simpler components that are individually designed, one at a time. The IP components are simply connected together in a graphical fashion to define the complete system. Shown here is part of a system defined in the Quartus Prime tool as an interconnection of separately developed IP blocks.